RECURSIVE SANDBOXES: EXTENDING SYSTRACE TO EMPOWER APPLICATIONS

Aleksey Kurchuk
Columbia University
ak2097@columbia.edu

Angelos D. Keromytis
Columbia University
angelos@cs.columbia.edu

Abstract The *systrace* system-call interposition mechanism has become a popular method for containing untrusted code through program-specific policies enforced by user-level daemons. We describe our extensions to *systrace* that allow sandboxed processes to further limit their children processes by issuing *dynamically* constructed policies. We discuss our extensions to the *systrace* daemon and the OpenBSD kernel, as well as a simple API for constructing simple policies. We present two separate implementations of our scheme, and compare their performance with the base *systrace* system. We show how our extensions can be used by processes such as *ftpd*, *sendmail*, and *sshd*.

1. Introduction

Systrace [Provos, 2003] has evolved to a powerful and flexible mechanism for containing untrusted code by examining system-call sequences issued by monitored processes. A user-level process assists the operating system kernel in determining whether a system call (and its associated arguments) are permitted by some externally-defined policy. This scheme allows unmodified programs to be effectively sandboxed and, combined with other convenient features (such as automatic policy generation and a graphical user interface), has made *systrace* widely used by OpenBSD [ope,] and NetBSD. However, the strength of the *systrace* approach — *i.e.*, keeping applications unaware of the sandbox — also makes the system less flexible than it could otherwise be.

Consider the case of a simplified *sendmail* daemon, which listens on the SMTP port for incoming connections, parses received messages, and delivers them to the recipient's mailbox. Typically, sendmail will invoke a separate

program to locally deliver a newly received email message. Different delivery programs may be invoked, based on parameters such as the recipient, *etc.* In a straightforward application of *systrace*, we would specify a policy that would limit filesystem accesses (*e.g.*, the *open()* system call) to files in the system mail-spool directory (*e.g.*, */var/spool/mail/*) and the sendmail configuration file. Compromise of the sendmail process, *e.g.*, through a buffer overflow attack, would restrict damage to these files and directories. However, if the local delivery program is susceptible to a buffer overflow attack when delivering to a particular user (*i.e.*, when using a particular local-delivery program), the subverted child process will be able to read or write to any user's mailbox and the configuration file.

More generally, static *systrace* policies do not allow the system to further limit its privileges based on information it has learned during its execution. A roughly similar example of such behavior is use of *seteuid()*, or of POSIX capabilities, where processes that do not need specific privileges can simply revoke them on themselves or their children processes.

We present an extension to *systrace* that allows sandbox-aware processes to communicate with their controlling daemon and specify further *restrictions* that can be placed on children processes of theirs. These policies can only **refine** the privileges already bestowed upon the sandboxed process — no "rights amplification" is possible. Processes that are aware of the sandbox can use a simple API to specify common restrictions they want to apply to their offspring. More elaborate restrictions can be specified by directly issuing *systrace* policies.

To ensure the monotonicity of privileges (*i.e.*, the fact that added policies do not expand the privileges of sandboxed processes), we explored two implementations. In the first approach, we used the KeyNote trust-management system [Blaze et al., 1999] to specify, parse, and evaluate policies in the *systrace* daemon. We modified the kernel to allow for indirect communication between the sandboxed process and the controlling *systrace* daemon, and modified the latter to generate and evaluate KeyNote policies. In the second approach, we used a recursive policy-evaluation model that emulates the KeyNote assertion evaluation process, which allowed the use of unmodified *systrace* policies at the cost of increased implementation complexity. Our benchmarks show that the added performance penalty of our extensions is negligible.

Paper Organization. The remainder of this paper is organized as follows. Section 2 discusses related work in process sandboxing and Section 3 gives an overview of the Systrace mechanism and KeyNote. Section 4 describes our extensions to Systrace, while Section 5 presents a preliminary performance evaluation of our extensions. Section 6 discusses our plans for future work, and Section 7 concludes this paper.

2. Related Work

The Flask system [Spencer et al., 2000] extends the idea of capabilities and access control lists by the more generic notion of a *security policy*. The Flask micro kernel system relies on a security server for policy decisions and on an object server for enforcement. Every object in the system has an associated security identifier, requests coming from objects are bound by the permissions associated with their security identifier. However Flask does not address the issue of cooperation amongst clients, servers and networks to deliver reliable and secure services to clients. Its notion of the security identifier is very limiting, in our system we require any number of conditions to hold before we provide a service, for example user identification might not be enough to grant access to a service, the user might also be required to access the service over a secure channel. As a minor issue, we have demonstrated that our prototype can be implemented as part of a widely used, commodity operating system, as opposed to a more fluid experimental micro-kernel.

System call interception, as used by systems such as TRON [Berman et al., 1995], MAPbox [Acharya and Raje, 2000], Software Wrappers [Fraser et al., 1999] and Janus [Goldberg et al., 1996] is an approach to restricting applications. TRON and Software Wrappers enforce capabilities by using system call wrappers compiled into the operating system kernel. The syscall table is modified to route control to the appropriate TRON wrapper for each system call. The wrappers are responsible for ensuring that the process that invoked the system call has the necessary permissions. The Janus and MAPbox systems implement a user-level system call interception mechanism. It is aimed at confining helper applications (such as those launched by Web browsers) so that they are restricted in their use of system calls. To accomplish this they use *ptrace(2)* and the */proc* file system, which allows their tracer to register a call-back that is executed whenever the tracee issues a system call. Other similar systems include Consh [Alexandrov et al., 1998], Mediating Connectors [Balzer and Goldman, 1999], SubDomain [Cowan et al., 2000] and others [Fraser et al., 1999, Ghormley et al., 1998, Walker et al., 1996, Mitchem et al., 1997].

Capabilities and access control lists are the most common mechanisms operating systems use for access control. Such mechanisms expand the UNIX security model and are implemented in several popular operating systems, such as Solaris and Windows NT [Custer, 1993]. The Hydra capability based operating system [Levin et al., 1975] separated its access control mechanisms from the definition of its security policy. Follow up operating system such as KeyKOS [Hardy, 1985, Rajunas et al., 1986] and EROS [Shapiro et al., 1999] divide a secure system into compartments. Communication between compartments is mediated by a reference monitor.

The methods that we mentioned so far rely on the operating system to provide a mechanism to enforce security. There are, however, approaches that rely on safe languages [Levy et al., 1998, Tardo and Valente, 1996, Leroy, 1995, Hicks et al., 1998], the most common example being Java [McGraw and Felten, 1997]. In Java applets, all accesses to unsafe operations must be approved by the security manager. The default restrictions prevent accesses to the disk and network connections to computers other than the server the applet was down-loaded from.

3. Overview of *systrace* and KeyNote

3.1 *systrace*

Systrace is a utility for monitoring and controlling an application's behavior by means of intercepting its system calls. Systrace provides facilities for confining multiple applications, interactive policy generation, intrusion detection and prevention, and can be used to generate audit logs. For a full account of how Systrace works, refer to [Provos, 2003]. Here, we give a brief review of the basic design of Systrace.

Taking a hybrid kernel/user-space approach, Systrace consists of a user-level daemon and a small addition to the OS kernel. Once a system call is issued by the monitored application, it is intercepted by the kernel part of Systrace, and is matched against an in-kernel policy. The in-kernel policy is a simple table of system calls and responses. If an entry for that system call exists, the result – allow or deny – is used in performing the operation. If no entry exists, such 'fast path' for the system call is impossible, and a user-level daemon is asked for a policy decision.

When the policy daemon receives a system call to be evaluated, it looks up the policy associated with the process, translates the system call arguments, and checks if the policy allows such a call.

Allowing for interactive policy generation, Systrace will prompt the human user for input on system calls not already described in the policy. If interactive policy generation is turned off, a system call that is not explicitly described in the policy will be denied.

3.2 KeyNote

KeyNote is a simple trust-management system and language developed to support a variety of applications. Although it is beyond the scope of this paper to give a complete tutorial or reference on KeyNote syntax and semantics (for which the reader is referred to [Blaze et al., 1999]), we review a few basic concepts to give the reader a taste of what is going on.

The basic service provided by the KeyNote system is *compliance checking;* that is, checking whether a proposed *action* conforms to local *policy.* Actions in KeyNote are specified as a set of name-value pairs, called an *Action Attribute Set.* Policies are written in the KeyNote *assertion language* and either accept or reject action attribute sets presented to it. Multiple assertions can be combined in specifying a policy, by constructing a delegation-graph, with each issuer of an assertion specifying the conditions under which the recipient of the assertion can perform some action. Ultimately, for a request to be approved, an assertion graph must be constructed between one or more POLICY assertions and one or more requesters. Because of the evaluation model, an assertion located somewhere in a delegation graph can effectively only refine (or pass on) the authorizations conferred on it by the previous assertions in the graph.

Each service that needs to mediate access, queries its local compliance checker on a per-request basis (what constitutes a "request" depends on the specific service and protocol). The compliance checker can be implemented as a library that is linked against every service or as a daemon that serves all processes in a host.

4. Extending *systrace*

In this section, we present two extensions to Systrace: nested policies and dynamic policy generation. Nested policies allow for a composition of several policies to be recursively applied to an application, and dynamic policy generation allows a Systrace-aware application to set an appropriate policy for itself and even to generate such policy at runtime.

4.1 Nested Policies

The motivation behind nested policies is that priveleges, granted to an application should be no greater than priveleges, enjoyed by its parent. We present two implementations of this idea, using different policy engines.

4.1.1 Systrace-KN. In our first attempt to introduce hierarchical policies to Systrace, we modified the user-level daemon to use KeyNote as its policy engine. Making use of KeyNote's native support for delegation of trust, we built an implicit graph of processes by making the parent process the 'Authorizer' of its offsprings' actions. This construction automatically restricts the privileges of the child process to those of its parent. Figure 1 shows a sample policy generated by Systrace-KN.

At run time, the 'Licensees' and 'Authorizer' fields are filled with the PIDs of the monitored process and its parent respectively. When a system call needs to be evaluated, its description is translated from the native Systrace format into a KeyNote request. The request is then evaluated by KeyNote, returning one of

```
KeyNote-version:  2
Authorizer:  "POLICY"
Licensees:  "systrace"
Conditions:
  (bin=="/bin/foo")->{
   (state=="default")->{
    (syscall=="fswrite")->{
     ((filename == "/usr/libexec/ld.so"));
     ((filename == "/var/run/ld.so.hints"));
     ...
    };
    ((syscall == "munmap"));
    ((syscall == "fsread") && (filename ==
"/<non-existent filename>:  /etc/malloc.conf"));
    ...
    ((syscall == "exit"));
   };
  };
```

Figure 1. Example of a policy, automatically generated by Systrace-KN.

the following values: {*ask*,*deny*,*permit*}. Stepping away from the recommendation in the KeyNote RFC, the first result - *ask* - is treated not as having the fewest permissions, but rather as a default one, which is returned if the request does not match the Action Attribute Set. If *ask* is returned in interactive mode, the user is asked for input. If the mode is automatic, the result is downgraded to *deny* just as in the original Systrace.

4.1.2 Systrace-H. Unsatisfactory performance of Systrace-KN (see Section 5) led us to extending the original Systrace policy engine to emulate support for nested policies. Following the idea of KeyNote, we define a notion of `compolicy` (complex policy). Each `compolicy` has a single `policy` (as defined in the original Systrace) as well as a link to an authorizing `compolicy`. Compliance with a `compolicy` is achieved by satisfying restrictions imposed by the associated policy and by all policies associated with each compolicy in the chain of authorizers. Effectively, this means that the system call, issued by a process, has to satisfy the policies of all of its ancestor processes.

A seemingly simpler solution of adding to the original Systrace policy a link to the authorizer proves not to be sufficient, since processes, associated with the same binary (and thus the same policy) might have a different set of ancestors. For example, consider program binaries `/bin/foo, /bin/bar,` and

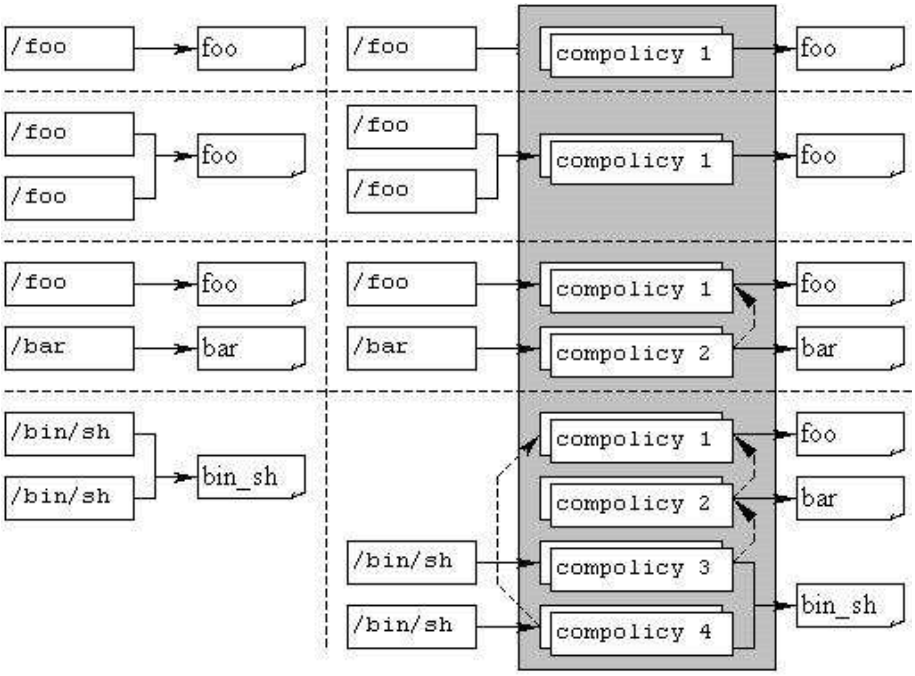

Figure 2. Side-by-side comparison of policy objects, created by original Systrace and those, created by Systrace-H.

`/bin/sh`, and the policies associated with them. The behavior of `/bin/foo` and `/bin/bar` is as follows:

/bin/foo:
```
fork()
child:  exec("/bin/bar")
parent: exec("/bin/sh")
```
/bin/bar:
```
exec("/bin/sh")
```

Running `/bin/foo` under Systrace will create the following policy chains: `foo`←`bar`←`sh` and `foo`←`sh`. Since the two processes are running `/bin/sh`, the behavior of the original systrace would be to use the same policy (bin_sh) for both of them. With hierarchical policies, this approach is incorrect since the privileges of one of the processes are restricted by policies of `/bin/foo` and `/bin/sh`, and the privileges of the other are further limited by policy `/bin/bar`.

Figure 2 illustrates the difference between the old approach and the one taken by our extension.

4.2 Run-time Policy Modification

In an effort to improve security, a Systrace-aware application might want to restrict its set of permissions to the minimum required by a particular code fragment. We present two approaches to run-time policy modification. The first one uses pre-generated policies, allowing the program to switch between them. The second one allows the program to modify (further restrict) its current Systrace policy with a dynamically generated one.

4.2.1 States. It is often convenient to view an application as a state machine. Being in different states, a program might naturally require different sets of privileges. For example, consider *sendmail*. While being in one state might require permission to establish network connections, these privileges are extraneous if the task at hand is to deliver a message to a local user's mailbox. The idea of narrowing the range of program's privileges is especially useful for further restricting the privileges of the children processes. Execution of a program always starts in *default* state.

State transitions. Naturally, not all state transitions will be valid. A subverted process should not be able to switch to an arbitrary state, aquiring excessive privileges. A separate policy is kept for rules associated with state switching. Every time an application requests to set a different state, this policy is queried. Instead of inventing a new policy engine for state transitions, we used the original Systrace policy engine. Each 'set state' operation is viewed as a system call *set* in emulation *state*, which has two possible arguments: *newstate* and *oldstate*.

File format. Using states required small changes to be made to the Systrace policy file name convention and to the file format itself. The state of the policy is used as an extension to the policy file name. For example, the policy file for `/bin/sh` in its default state should be named `bin_sh.default`. State is also reflected in the first line of the policy file as in the following example:

```
[ bin_sh.default ]
Policy: /bin/sh, Emulation: native, State: default
```

The state policies are written in the same language used for creating regular Systrace policies, and are stored in files with extension *state*. Thus, *state* SHOULD NOT be used as a state identifier. Also, state names that start with an underscore are reserved for use by Systrace as explained in Section 4.2.2. Consider the following code snippet in an application:

```
fork();
if (child) {
   systrace_setstate("risky");
   do_risky();
   ...
}
else if (parent) {
   ...
}
```

Here, the state of the child process is switched to *risky*, possibly restricting the range of actions the process can perform. Note also that since the state transition policy below does not explicitly allow a `risky`→`default` transition, a rogue process will not be able to change its state back to *default* once it enters the *risky* state.

```
[ bin_foo.state ]
Policy:  /bin/foo, Emulation:  state, State:  state
state-set:  oldstate eq "default" and newstate eq "risky"
            then permit
```

Note that when switching states, the policy, associated with the new state does not restrict the old one, but replaces it, as shown in Figure 3. The mechanism for restricting the existing policy is discussed in the next section.

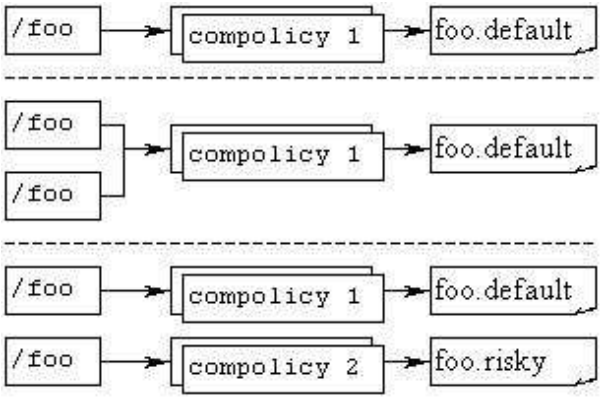

Figure 3. Policies after running the code snippet changing state to `risky`.

4.2.2 Dynamic Policy Creation. While state switching allows applications to choose the most suitable (but static) policy, one might want to

have a policy based on information acquired during the program's execution. Dynamic policy creation allows one to refine an application's privileges by providing run-time generated policies.

In order to facilitate creation of dynamic policies, a (pseudo) system call *all* was introduced, which allows a policy to specify restrictions, applying the action, associated with *all*, to all system calls not explicitly described. If a filter for *all* is not specified explicitly, it is taken to be *deny* by default. Consider, for example, how sendmail would make use of this functionality when delivering local mail to user `jdoe`. Let variable `restricted_policy` contain the following policy, generated at run time:

```
native-fsread:  filename sub "/var/mqueue/jdoe/" then permit
native-fsread:  filename sub "/var/mqueue/" then deny
...
native-all:  permit
```

Since policy filters are matched in top-down order, reading from directory `jdoe` is allowed, while an attempt to open a file in someone else's directory will be thwarted. The following code will run the local delivery agent with restricted privileges, allowing it to access specific user's mail queue directory, but not that of any other user:

```
systrace_setpolicy(restricted_policy);
exec(local_delivery_agent,...);
```

Once the Systrace daemon receives a set_policy request, it creates a new `policy` object. This policy's state starts with an underscore, indicating that this is a volatile policy. Such policies do not get saved to disk. Thus, user defined state identifiers should not start with an underscore.

4.2.3 Communication. In order for run-time policy modification to work, there needs to be a way for the controlled application to communicate with the Systrace daemon. We briefly describe how this interaction is accomplished.

To send a message to the user-level Systrace daemon, a Systrace-aware application sends a message to */dev/systrace*, which in turn relays the message to the appropriate Systrace daemon process. The messages are encoded with the following data structure:

```
struct systrace_message {
   unsigned int processed;
   unsigned int code;
   size_t       length;
```

```
    char         message[1];
};
```

Variable `length` specifies the length of the `message`, and `code` identifies the action requested. So far, the only relay codes defined are `SYSTRACE_RELAY_SETSTATE` and `SYSTRACE_RELAY_SETPOLICY`. Structure `systrace_relay` is then populated and sent to `/dev/systrace` via the `SYSTR_RELAY ioctl()`.

```
struct systrace_relay {
    pid_t    pid;
    int      flag;
    key_t    key;
    size_t   size;
};
```

In the `systrace_relay` structure, `pid` is the process ID of the monitored process, `flag` identifies the requested action, `key` is the key of the shared memory region of size `size`.

Having sent the message, the application waits for flag `processed` to be set. The response from the daemon is stored in `code`. The following responses are currently defined:

`SYSTRACE_RELAY_SUCCESS` - Requested operation completed successfully
`SYSTRACE_RELAY_FAILURE` - Unexpected failure
`SYSTRACE_RELAY_PERMISSION` - Application is unauthorized to perform this operation. This error may occur if, for example, an invalid state change is requested.
`SYSTRACE_RELAY_ARGUMENT` - Invalid argument specified. An invalid Systrace policy passed to `systrace_setpolicy()` might cause this error message.

The kernel, upon receipt of the message, signals the Systrace daemon that a message is waiting for it. The daemon picks up the pending relay via a `STRIOCRELAY ioctl()`. When the message is processed, daemon sets `code` to one of the above responses, and the `processed` flag to 1.

5. Performance Evaluation

The policy enforcement agent communicates with the policy engine via three callback functions. These functions are responsible for creating state for a new process (EXEC), evaluating a system call with known (translated) arguments (TRANS), and general calls with no (or not translated) arguments (GEN). In order to measure performance of different versions of Systrace, we

timed these three functions. As a test application, we used the `sendmail` daemon, making necessary modifications for it to use policy modification features.

The following table shows the average times (in micro-seconds) needed for each operation. 'Systrace-H (states)' denotes times, observed when running sendmail, modified to have several states. In 'Systrace-H (dynamic)', sendmail was changed to generate a dynamic policy for delivering local mail (as in the example in Section 4.2.2).

	GEN	TRANS	EXEC
Original Systrace	2	37	54
Systrace-KN	902	866	173
Systrace-H	20	37	81
Systrace-H (states)	20	37	89
Systrace-H (dynamic)	20	37	121

One obvious result of these performance measurements is that Systrace-KN is unacceptably slow. Further investigation proved that over 99% of the time was spent in $kn\_query()$ function — the KeyNote function that evaluates a request. This moved us to turn away from a KeyNote-based implementation, but use the ideas of KeyNote to enhance the policy engine of the original Systrace.

In all variations of Systrace-H, the time needed to evaluate a translated system call remained the same as in the original Systrace. The original Systrace policy engine is very lightweight, and more precise measurements would be needed to detect the difference. The time to evaluate a call with no arguments, however, has risen, almost reaching that of TRANS.

The reason for such a slowdown is as follows. In the original Systrace, upon creation of a new process, all general system calls were entered in the kernel policy (pre-loaded) to speed up the execution and to take the responsibility for their evaluation off the user-level Systrace daemon. Thus, all general system calls that needed to be evaluated by the user-level daemon were considered non-compliant with the policy. Such pre-loading is not done in Systrace-H, since it requires all ancestor policies to be queried at the start of the process's execution. Instead, a general system call is evaluated the same way a translated one is (explaining the increase in evaluation time), and the kernel-policy is modified, reflecting the result of the evaluation. Since such evaluation is done once per call per process, the performance degradation is insignificant.

The increase in time when evaluating EXEC is most likely due to the fact that in Systrace-H, there generally exist more `compolicy` objects to search, and the fact that aside from loading the default policy, the state transition policy (if one exists) needs to be loaded as well. Since 'exec' calls are relatively infrequent, this increase results in negligible performance penalty.

6. Future Work

Having established a mechanism for a controlled application to communicate with the controlling Systrace daemon, we used it to relay requests for policy changes. In the future, the same mechanism may be used to perform a wider variety of tasks, including communication with other processes under Systrace's control.

Lack of tools for programmatically creating Systrace policies might impede the use of dynamically generated policies. Development of tools to facilitate policy generation should greatly decrease the time needed to allow a Systrace-aware application to make use of these features. Such work is in our current work plans.

Finally, we plan to extend the current implementation of Systrace-H to allow pre-loading, per our discussion of possible performance degradation in Section 5.

7. Conclusions

This paper presents an extension to the Systrace facility for process sandboxing. We argue that using hierarchical policies adds security, while having an acceptable performance overhead. Our mechanism adds the ability to control the behavior of all binaries run by a single aplication by modifying a single policy.

We further showed that an application may enhance its security by using state policies and the ability to dynamically generate policies. The changes necessary to transform an application into a Systrace-aware one are minimal and can be made in minutes.

From our perliminary performance evaluation we conclude that a policy engine as generic and flexible as KeyNote might prove to be too slow for some tasks. A highly specialized policy engine that follows the same evaluation model might be called for instead.

Acknowledgements

This work was supported by NSF under Contract CCR-TC-0208972.

References

[ope,] The OpenBSD Operating System. `http://www.openbsd.org/`.

[Acharya and Raje, 2000] Acharya, Anurag and Raje, Mandar (2000). Mapbox: Using parameterized behavior classes to confine applications. In *Proceedings of the USENIX Security Symposium*, pages 1–17.

[Alexandrov et al., 1998] Alexandrov, A., Kmiec, P., and Schauser, K. (1998). Consh: A confined execution environment for internet computations.

[Balzer and Goldman, 1999] Balzer, Robert and Goldman, Neil (1999). Mediating connectors: A non-bypassable process wrapping technology. In *Proceeding of the 19th IEEE International Conference on Distributed Computing Systems*.

[Berman et al., 1995] Berman, Andrew, Bourassa, Virgil, and Selberg, Erik (1995). TRON: Process-Specific File Protection for the UNIX Operating System. In *Proceedings of the USENIX Technical Conference*.

[Blaze et al., 1999] Blaze, M., Feigenbaum, J., Ioannidis, J., and Keromytis, A. D. (1999). The KeyNote Trust Management System Version 2. RFC 2704.

[Cowan et al., 2000] Cowan, Crispin, Beattie, Steve, Pu, Calton, Wagle, Perry, and Gligor, Virgil (2000). SubDomain: Parsimonious Security for Server Appliances. In *Proceedings of the 14th USENIX System Administration Conference*.

[Custer, 1993] Custer, Helen (1993). *Inside Windows NT*. Microsoft Press.

[Fraser et al., 1999] Fraser, Tim, Badger, Lee, and Feldman, Mark (1999). Hardening COTS Software with Generic Software Wrappers. In *Proceedings of the IEEE Symposium on Security and Privacy*.

[Ghormley et al., 1998] Ghormley, Douglas P., Petrou, David, Rodrigues, Steven H., and Anderson, Thomas E. (1998). SLIC: An Extensibility System for Commodity Operating Systems. In *Proceedings of the USENIX Technical Conference*, pages 39–52.

[Goldberg et al., 1996] Goldberg, Ian, Wagner, David, Thomas, Randi, and Brewer, Eric A. (1996). A Secure Environment for Untrusted Helper Applications. In *Procedings of the USENIX Technical Conference*.

[Hardy, 1985] Hardy, Norman (1985). The KeyKOS. *Operating Systems Review*, 19(4):8–25.

[Hicks et al., 1998] Hicks, M., Kakkar, P., Moore, J. T., Gunter, C. A., and Nettles, S. (1998). PLAN: A Programming Language for Active Networks. Technical Report MS-CIS-98-25, Department of Computer and Information Science, University of Pennsylvania.

[Leroy, 1995] Leroy, X. (1995). Le système Caml Special Light: modules et compilation efficace en Caml. Research report 2721, INRIA.

[Levin et al., 1975] Levin, R., Cohen, E., Corwin, W., and Wulf, W. (1975). Policy/mechanism separation in hydra. In *Proceedings of the 5th ACM Symposium on Operating Systems Principles*, pages 132–140.

[Levy et al., 1998] Levy, Jacob Y., Demailly, Laurent, Ousterhout, John K., and Welch, Brent B. (1998). The Safe-Tcl Security Model. In *Proceedings of the USENIX Technical Conference*.

[McGraw and Felten, 1997] McGraw, Gary and Felten, Edward W. (1997). *Java Security: hostile applets, holes and antidotes*. Wiley, New York, NY.

[Mitchem et al., 1997] Mitchem, T., Lu, R., and O'Brien, R. (1997). Using Kernel Hypervisors to Secure Applications. In *Proceedings of the Annual Computer Security Applications Conference*.

[Provos, 2003] Provos, N. (2003). Improving Host Security with System Call Policies. In *Proceedings of the 12th USENIX Security Symposium*.

[Rajunas et al., 1986] Rajunas, S.A., Hardy, N., Bomberger, A.C., Frantz, W.S., and Landau, C.R. (1986). Security in KeyKOS. In *Proceedings of the IEEE Symposium on Security and Privacy*.

[Shapiro et al., 1999] Shapiro, Jonathan S., Smith, Jonathan M., and Farber, David J. (1999). EROS: a fast capability system. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pages 170–185.

[Spencer et al., 2000] Spencer, R., Smalley, S., Loscocco, P., Hibler, M., Anderson, D., and Lepreau, J. (2000). The flask security architecture: System support for diverse security policies. In *Proceedings of the USENIX Security Symposium*, pages 123–139.

[Tardo and Valente, 1996] Tardo, J. and Valente, L. (1996). Mobile Agent Security and Telescript. In *Proceedings of the 41st IEEE Computer Society Conference (COMPCON)*, pages 58–63.

[Walker et al., 1996] Walker, K. M., Stern, D. F., Badger, L., Oosendorp, K. A., Petkac, M. J., and Sherman, D. L. (1996). Confining root programs with domain and type enforcement. In *Proceedings of the USENIX Security Symposium*, pages 21–36.